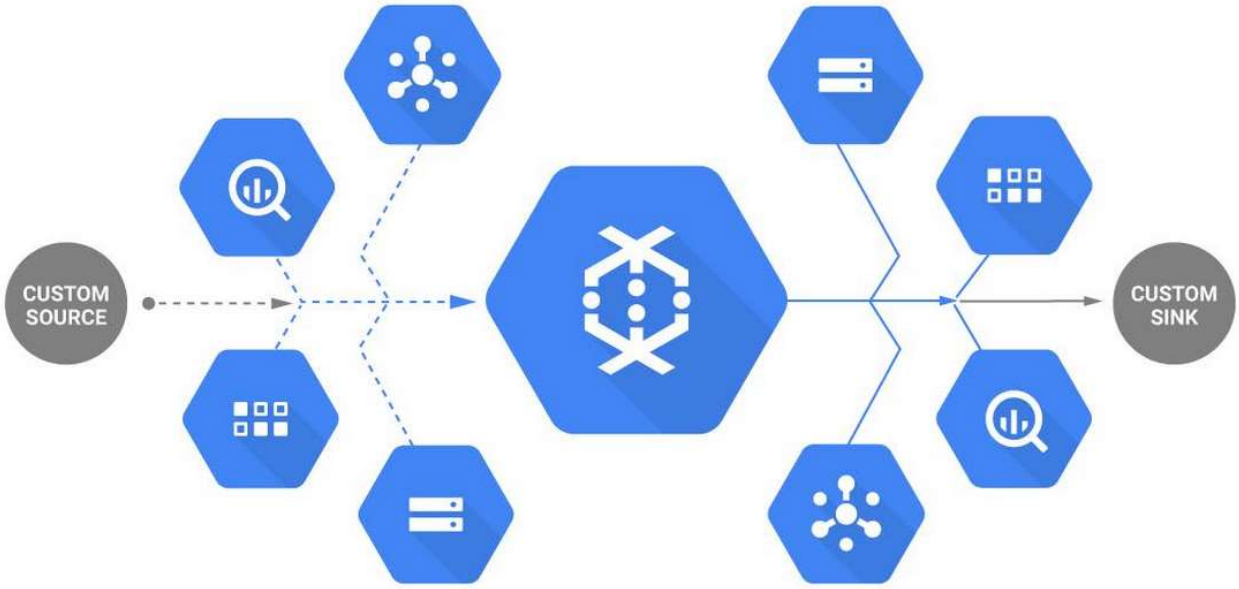
Libertando dados massivos presos em uma aplicação web

Leonardo Miguel
Jan 10, 2019 · 11 min read



Grande parte do sucesso do Arquivei como produto vem do valor dos dados de seus clientes. A Nota Fiscal Eletrônica (NFe) por si só possui mais de 700 campos contendo informações diversas de transações comerciais, das empresas envolvidas, de produtos, além de intermináveis informações tributárias. A análise dessas informações é capaz de trazer imenso valor para as empresas e para a Arquivei, que as usa para entender melhor seus clientes e como eles podem usufruir mais desses dados.

Onde estão os dados?

Neste post trataremos especificamente do problema das Notas Fiscais Eletrônicas. Uma NFe é um documento que deve ser emitido para o fisco a cada transação financeira de uma empresa, de forma que as empresas envolvidas estejam em dia com suas obrigações fiscais. A Arquivei consulta junto da Secretaria da Fazenda (Sefaz) as informações das NFes emitidas contra seus clientes através de seus certificados digitais.

Cada NFe possui um XML associado onde estão contidas suas informações, que tem em média 5 kilobytes. A Arquivei armazena cada um desses documentos no AWS S3, sendo que cada um dos milhões de documentos que compõem a base de dados da Arquivei hoje está em um arquivo diferente. Além disso, o banco de dados da aplicação armazena informações próprias para busca em um cluster de Elasticsearch, que serve as partes mais analíticas da aplicação hoje.

Toda essa base de dados representa hoje aproximadamente 5 terabytes de dados, se consideradas as replicações e metadados. Isso considerando apenas um tipo de documento fiscal, a NFe. A Arquivei trabalha ainda com mais 4 outros documentos fiscais e seus respectivos eventos de manifestação e cancelamento que possuem suas devidas peculiaridades também.

O desafio

O time de Engenharia de Dados da Arquivei é responsável por libertar os dados e disponibilizá-los para análise nos mais diversos níveis, assegurando sua consistência, e levando em conta a criticidade, confidencialidade, e segurança dos dados. Dentro da vocação de ser uma empresa cada vez mais *data-driven*, o time vem crescendo e cada vez mais demandas chegam para geração de relatórios e novos produtos sobre documentos fiscais.

Assim, nasceu o projeto de unificar em um mesmo *dataset* todas as informações de NFes disponíveis na Arquivei. O objetivo inicial é extrair todos os dados, tanto dos XMLs armazenados no S3 quanto do banco de dados da aplicação, e disponibilizá-los no Data Warehouse da empresa.

A grande restrição é que não podíamos utilizar diretamente o banco de dados da aplicação para nossas análises porque impactaria o ambiente de produção, sendo que poderia causar lentidão ao cliente. O Elasticsearch é muito poderoso, sendo capaz de fazer agregações e entregar resultados em poucos segundos, porém a forma como nossa aplicação é pensada não engloba o tipo de requisição que precisamos fazer.

Um cliente é capaz no Arquivei de gerar relatórios em cima dos dados de sua empresa, e mesmo sendo um cliente grande é esperado que o banco de dados seja capaz de entregar esses relatórios. Porém, quando falamos de análise de documentos fiscais estamos falando de utilizar muito mais dados, muitas vezes o filtro não é por empresa mas por região geográfica, categoria de produto, ou tipo

de cliente. Ou seja, o tipo de *query* que fazemos não consegue aproveitar a estrutura de índices da aplicação existente hoje, e o trabalho de mudar isso tudo seria enorme.

Arquitetura de Dados

A maior parte da aplicação Arquivei roda hoje utilizando os serviços da Amazon Web Services (AWS), onde o time de Engenharia de Dados criou sua primeira arquitetura de dados. Sua stack era composta de: Apache Airflow, Amazon Redshift, e Python (usando algumas libs de ETL com paralelismo). Devido à problemas de escalabilidade e custo inicial, passamos a estudar alternativas fora da AWS. O time à época era muito pequeno e havia uma preocupação com a mão de obra necessária para usar tecnologias tradicionais como Hadoop.

O time migrou então para uma stack montada usando os serviços da Google Cloud Platform (GCP), inspirada em uma série de artigos do Spotify, composta por: Google Pub/Sub, Google Dataflow, e Google BigQuery. Nesta mesma época, apresentamos o conceito de Arquitetura de Eventos para os outros times que faziam parte do departamento de Engenharia, usando inicialmente para coleta de dados com pouca intrusão na aplicação. Essa arquitetura passou por algumas modificações e hoje usamos algo parecido, tendo substituído o Google Pub/Sub pelo Apache Kafka, por razões que podemos comentar em outro post.

Um dos desafios então de libertar os dados de NFes está justamente em estarmos operando em uma *cloud* diferente do restante dos sistemas da empresa, o que adiciona problemas de latência e custos de rede, além de ter que lidar com algumas incompatibilidades e diferenças.

Apache Kafka

Nosso sistema de distribuição de mensagens hoje é o Apache Kafka. Nós mantemos um cluster do mesmo na AWS por questões de latência da aplicação — você pode aprender como é feito nosso *setup* aqui. O Kafka é responsável por receber mensagens enviadas da aplicação (escrita em PHP). Essas mensagens possuem o seguinte formato JSON:

```
{
    "ID": "01CXAYNVB2YPEFNC7RC1VPQUND",
    "SchemaVersion": 1,
    "DataVersion": 1,
    "Source": "app",
    "Type": "event-was-sent",
    "CreatedAt": "2018-11-27T14:05:18-02:00",
    "Data": {
        "Key1": "Value1",
        "Key2": "Value2"
    }
}
```

Utilizamos como ID o padrão Universally Unique Lexicographically Sortable (ULID). SchemaVersion determina a versão do envelope de eventos, e DataVersion a versão dos campos que virão dentro de Data. Nós prezamos para que os eventos sejam retrocompatíveis, mas isso nem sempre é possível. Source indica qual sistema enviou um determinado evento, e Type indica qual o tipo de evento enviado (pedimos para que sempre represente uma ação com um verbo no passado). CreatedAt é a data de criação do evento (quando ocorreu a ação) no formato RFC 3339, onde pedimos a maior precisão possível na fração de segundo. Por fim, o campo Data é um campo onde o produtor pode colocar qualquer campo que seja útil ao evento sendo enviado, sendo que sempre que isso for modificado ele deve alterar o DataVersion para manter consistência.

Os nossos requisitos para um bom formato de mensagem são:

- o evento deve ser univocamente identificável (através do ID)

- o evento deve ser ordenável — deve ser possível verificar a ordem em que dois eventos foram produzidos (através do ULID e/ou CreatedAt)
- o evento deve representar um fato, algo que com certeza aconteceu em determinado ponto no tempo

Além disso, um evento pode ser de um dos seguintes tipos (segundo este belíssimo post do Martin Fowler):

- event notification: evento que notifica um sistema de que algo aconteceu, geralmente sem carregar muitas informações sobre o que aconteceu
- event-carried state transfer: evento usado para manter um consumidor atualizado sobre um determinado estado, enviando a maior quantidade de informações possíveis para poupar seu trabalho, carregando portanto uma grande quantidade de informações
- event-sourcing: evento enviado de forma que a partir dele seja possível reconstruir o estado de qualquer sistema em qualquer ponto no tempo — aqui o evento é a fonte de verdade, enquanto nos outros tipos estamos apenas reportando o estado do banco de dados, onde se encontra a fonte de verdade deles

Tudo isso, com o menor overhead possível (e para isso JSON não é um bom padrão, sendo que sabemos de alternativas melhores como Avro e Protobuf).

Google Dataflow

O serviço que utilizamos para processar bases de dados massivas é o serviço gerenciado da Google, o Dataflow. Este serviço é um dos grandes motivos de termos migrado para a GCP. Nele, utilizamos o framework Apache Beam (projeto desmembrado do Google Dataflow) capaz de simplificar a lógica de processamento destes dados. Usando Dataflow com Beam, tiramos vantagem das diversas *features* do Beam com relação a conectores I/O, operadores de junção e transformação típicas de ETL, além da tolerância a falha e *autoscaling* fornecidos pelo Dataflow. Outra lib que facilita bastante nossa vida é a Spotify Scio, que porta as funcionalidades do Beam de Java para Scala porque ninguém merece codar em Java né gente.

O Dataflow possui conectores prontos para o Apache Kafka, diversos bancos de dados, e sistemas de mensagens. Através deles, nós lemos as mensagens enviadas pela aplicação, os transformamos de forma facilitada pelo modelo do Beam, e os enviamos para nosso Data Warehouse.

Google BigQuery

Outro serviço que facilita muito nossa vida é o Google BigQuery. Por ser um banco colunar analítico gerenciado, temos uma preocupação mínima sobre como gerenciar os dados lá armazenados, não precisando gerenciar uma infraestrurura própria para armazenamento destes dados. O fato de ser extremamente escalável também é um ponto bastante positivo pra o tipo de operação que estamos querendo fazer.

O serviço para nós tinha um custo muito menor que manter um cluster de Redshift. O custo de armazenamento é bem reduzido e próximo ao de serviços como GCS (aproximadamente $0.02/GB), com os benefícios automáticos de compactação nativa e do *long-term storage* — se você não tocar em um dado por 90 dias, o custo de armazemaneto cai 50% sem perca de performance. Já no custo de *queries* são considerados apenas os dados tocados, sendo que se você referencia uma coluna em uma *query*, é cobrado o tamanho inteiro da coluna, por exemplo — custo esse otimizável através de boas práticas utilizadas pelos times de análise. O desempenho do BigQuery para *queries* enormes é incrivelmente escalável, levando alguns segundos para fazer operações de JOIN complexas, operações utilizando funções janela, e até mesmo deduplicações. Recomendo muito a leitura de como funciona o BigQuery.

O tamanho do problema

Nossa primeira abordagem para tentar recuperar todos os documentos do S3 foi fazer um *job* no Dataflow que lesse diretamente desse serviço, fizesse o *parsing* do XML desses documentos, e os enviasse para o Data Warehouse. A primeira limitação que encontramos foi que não conseguimos escalar a quantidade de requisições ao S3, onde conseguimos ler 1~2 milhares de documentos por segundo mesmo com um número bastante alto de *workers*.

Existem alguns motivos para não conseguirmos escalar além disso:

- O *bucket* que utilizamos para leitura é o mesmo que é acessado pela aplicação, sendo que concorríamos com a quantidade de *requests* da aplicação, chegando próximos do limite de requisições ao S3 por segundo.
- A latência entre a rede da GCP e o S3 era um pouco alta, sendo que precisávamos de pelo menos duas requests para recuperar cada XML, além do tempo de download do documento em si.

A essa taxa, levaríamos dias para processar toda a nossa base de dados, e a cada reprocessamento teríamos que perder mais dias de trabalho para monitorar a execução dos jobs, além do custo bastante alto para nossa realidade. Uma das maiores vantagens do Dataflow é sua capacidade de usar uma grande quantidade de *workers* para processar conjuntos massivos de dados sem que tenhamos a preocupação de gerenciar essa infraestrutura. Nossa maior preocupação nesses casos é o custo, mas é comum que tenhamos problemas ao extrair uma grande quantidade de dados devido aos limites de nossas fontes de dados.

Além disso, os documentos não são armazenados de forma incremental, ou seja, não é fácil saber quais os XMLs que foram criados em uma determinada faixa de tempo olhando apenas para o banco da aplicação. Nossa primeira tentativa de processar toda a base de dados histórica da Arquivei, levou mais de 20 horas rodando em 80 máquinas devido a essas restrições, representando um custo alto e inviável.

A solução

Para solucionar o problema da alta quantidade de acessos ao S3 e para possibilitar o rápido reprocessamento dos dados a um custo viável, criamos o conceito de *bundle*. Um bundle é um pacote de XMLs criados no banco da aplicação em uma determinado hora (essa granularidade pode ser ajustada). Esse bundles são armazenados no Google Cloud Storage em formato Avro.

Escolhemos armazenar os bundles no GCS devido a sua maior flexibilidade em relação ao tamanho dos objetos (se comparado com o BigQuery), sua escalabilidade, e a integração nativa com o BigQuery através das tabelas externas. O formato Avro foi escolhido por reduzir os custos de armazenamento (através da compressão nativa), por ter fácil integração com Java e Scala, e permitir a imposição de um determinado schema — um problema comum quando se lida com JSON.

Uma característica importante dos *bundles* é que eles devem ser o mais simples possível, para evitar que tenhamos que recriá-los sempre por problemas de consistência ou manutenção. Sendo assim, incluímos nos *bundles* apenas as informações que sabíamos que eram imutáveis.

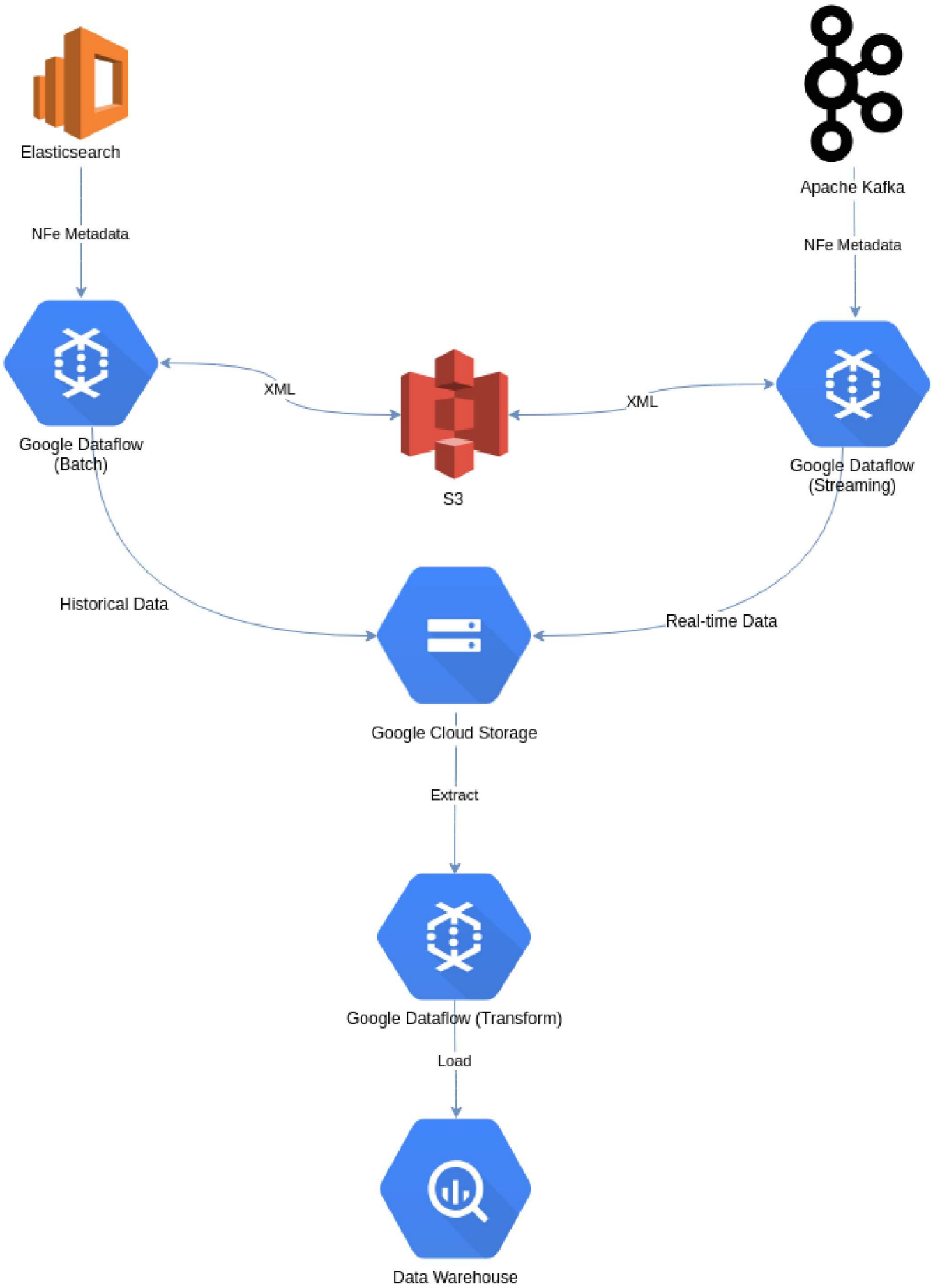
Criamos então dois jobs a partir dessa ideia:

- um *batch* histórico que lê os caminhos dos documentos de um snapshot retirado do Elasticsearch da aplicação para recuperar os documentos do S3 e os armazenar nos bundles. Introduzimos o Elasticsearch para reduzir os tempos de listagem de documentos do S3 (que tomava tempo significativo no job original), assim já sabemos o caminho de cada documento no S3, economizando uma request para cada documento.

- um *batch* diário que processa os dados conforme os documentos são enviados pela aplicação para o Kafka, e os processa armazenando em bundles. A princípio pensamos em utilizar *streaming* para esses dados, mas por questões de custo e simplificação fizemos a versão com um batch que recupera os dados do dia anterior.

Os bundles armazenados em Avro no S3 são lidos por jobs diários, processados (etapa em fazemos o *parsing* do XML), e armazenados no BigQuery. O fato de usarmos bundles, possibilita o processamento incremental, uma vez que neles está contida a informação de data de criação.

A arquitetura ideal aqui descrita é ilustrada na imagem abaixo:



Parsing

Uma etapa interessante desse processo foi lidar com o *parsing* dos XMLs das NFes. Nossa aplicação já contém um *parser* utilizado para os produtos e para formatação dos dados a serem inseridos no banco de dados. Porém, hoje esse parser está escrito em Golang não sendo diretamente viável utilizá-lo em nossos códigos.

A primeira solução para este problema foi criar um microserviço de *parsing* rodando em um Kubernetes, que era acessado pelos jobs via HTTP. O problema dessa solução foi o trabalho manual de escalar a quantidade de pods toda vez que tínhamos que rodar algo maior, além de problemas de conexão causados pelo protocolo que estávamos usando quando mantínhamos um número muito grande de conexões.

Para não termos o retrabalho de criar em Java ou Scala um parser que tenha que lidar com todas as peculiaridades das diferentes versões de NFe, conseguimos integrar o binário gerado pela equipe responsável pelo *parser*, através da biblioteca Java Native Access (JNA), que é um assunto legal para outro post.

Resultados

O job histórico para criar os bundles teve desempenho semelhante ao citado anteriormente para processar todos os XMLs diretamente do S3, levando pouco mais de 20 horas para ser executado, causando um custo razoável.

Os ganhos vieram a seguir: com os bundles somos capazes de processar todos os XMLs do Arquivei em pouco mais de uma hora usando *autoscaling* do Dataflow, sendo que o processamento dos dados de um dia leva cerca de 10 a 20 minutos (em grande parte por conta do tempo de *warmup*).

Através de tabelas externas, conseguimos ainda fazer análises preliminares diretamente sobre os bundles, que pode ser utilizada para contagens e relatórios com menor latência de dados.

Problemas encontrados

Devido a algumas características próprias da aplicação, o tamanho dos *bundles* é significativamente irregular, com tamanhos que variam de alguns kilobytes a 500 megabytes. Isso dificulta o trabalho de *autoscaling* do Dataflow e também causou problemas de consumo de memória ao processar uma grande quantidade de arquivos grandes.

Também não criamos nenhuma rotina para garantir unicidade de informações, então em alguns casos pode ocorrer duplicação de XMLs entre bundles o que dificulta processamentos posteriores.

Próximos passos

Para otimizar e aperfeiçoar o conceito de *bundles*, pretendemos criar jobs que realizam a manutenção dos mesmos, padronizando tamanho, tratando duplicações, e verificando sua consistência.

Espero ter conseguido mostrar um pouco de como funciona nossa infraestrutura de dados e como conseguimos montar o Data Warehouse da Arquivei usando as ferramentas certas e otimizando para nossos casos de uso.